Fault Tolerance via Replication in Coarse Grain Data-Flow[1]

Anh Nguyen-Tuong, Andrew S. Grimshaw and John F. Karpovich

University of Virginia, Thornton Hall, Department of Computer Science
Charlottesville, VA 22903

Email: {nguyen | grimshaw | karp}@virginia.edu

URL: http://www.cs.virginia.edu/~an7s, http://www.cs.virginia.edu/~grimshaw, http://www.cs.virginia.edu/~jfk3w

1. Introduction

Recent advances in network technology promise to make gigabit-per-second bandwidth between remote hosts a reality in the near future. This increase in bandwidth paves the way for increased exploitation of distributed computing resources. Coupled with advances in distributed memory parallel compiler technology, there is strong reason to believe that wide-area distributed parallel processing will be an increasingly popular and important programming paradigm. Parallelizing and distributing program sub-tasks has the potential to increase performance for many applications while also improving the overall utilization of system resources. Unfortunately, there is a downside. When a program is partitioned into sub-tasks, each sub-task is distributed to potentially a different processor. As the number of processors employed by an application increases so does the chance that the application will fail due to a host/processor failure.

At the University of Virginia, we have experienced first hand the problems caused by host failures in distributed systems while developing and using a prototype for the Legion project[2] [13][14]. The objective of Legion is to construct the software environment to enable a nation-wide or world-wide virtual computer capable of supporting distributed and parallel applications. Our current prototype, which we call the Campus-Wide Virtual Computer (CWVC), contains a mix of over 90 workstations and an IBM SP-2 multicomputer. Even in this relatively small environment, we are frequently experiencing host failures. On the scale of the envisioned nation-wide system, host failures will simply be a fact of life and must be dealt with accordingly. User applications, especially those that are critical or are composed of many distributed components, must be resilient to host failures. Fortunately developing fault tolerant parallel applications does not need to be difficult.

In this paper we show that by developing applications using the data-flow model of parallel computation there is a simple method for providing fault-tolerance. The key to our approach is in exploiting the functional nature of data-flow programs in the fault-

1. This work is partially funded by NSF grants ASC-9201822, NRaD contract N00014-94-1-0882, and ARPA grant J-FBI-93-116.

2. Information on Legion is available on the WWW at http://www.cs.virginia.edu/~legion

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**
4. TITLE AND SUBTITLE **Fault tolerance via Replication in Coarse Grain Data-Flow**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Cahrlottesville,VA,22094-4740**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**16**	

tolerance mechanisms. Recall that data-flow computations are modelled by actors, arcs, and tokens. Actors are computation primitives, tokens carry data or control information, and arcs are used to model the dependencies between actors. The distinguishing feature of actors in terms of fault tolerance is their idempotent nature: an actor presented with the same tokens will produce the same result. Thus, fault-tolerance can be easily achieved through actor replication, i.e. replicate an actor *k* times and use the first available result (discard later arriving results).

To illustrate this approach, we implemented automatic actor replication within the Mentat system [9][12]. Mentat is an object-oriented, data-flow based, parallel processing system. We used a synthetic pipeline program as a simple application to demonstrate the power of the approach and to explore the trade-offs between fault-tolerance, performance and resource consumption. Resource consumption is often neglected but must be taken into account as we move towards an environment in which machines are shared. We plan to expand our results in the future using applications already in use on the CWVC, e.g. DNA sequence comparison, VLSI routing, to name just a few.

The remainder of the paper is organized as follows. We first present a brief overview of the Mentat system and its execution model (Section 2). We describe the protocol used to transparently replicate actors and illustrate the process of converting Mentat source code to an actual run-time implementation (Section 3). We show experimental data taken on a synthetic pipeline application using our transparent replication method (Section 4) and analyze the results (Section 5). We then present an optimization to our protocol which will be included in the next release of Mentat (Section 6). Finally, we discuss related work (Section 7) and conclude (Section 8).

2. Mentat

Mentat[3] is a high performance, object-oriented parallel processing system. There are two primary aspects of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system. MPL is an object-oriented programming language based on C++. The granule of computation is the Mentat class member function. The programmer is responsible for identifying those object classes whose member functions are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used like C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the environment.

The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention.

Mentat classes are denoted by the inclusion of the keyword “mentat” in the class definition, as in the mentat class `sw_worker` shown in Figure 1. The keyword `mentat` tells the compiler that the member functions of the class are worth executing

3. Information on Mentat is available on the WWW at http://www.cs.virginia.edu/~mentat

in parallel. Mentat classes may be defined as either *persistent* or *regular*. Instances of regular Mentat classes are logically stateless, thus the implementation may create a new instance to handle every member function invocation. Persistent Mentat classes maintain state information between member function invocations. This is an advantage for operations that require large amounts of data, or that require persistent semantics.

Figure 1 Mentat class definition

```
regular mentat class sw_worker {
// private data and function members
public:
      result_list*compare(sequence*,libstruct*,paramstruct);
};
```

The keyword “mentat” tells the compiler to treat instances of this class differently. The “regular” modifier indicates that instances of this class are stateless, i.e., they are pure functions.

A Mentat object is an instance of a Mentat class, and possesses a name, a thread of control, and an address space. Because Mentat objects each have their own address space they are address space disjoint. Therefore, *all* communication between Mentat objects and between Mentat objects and main programs is via member function invocation.

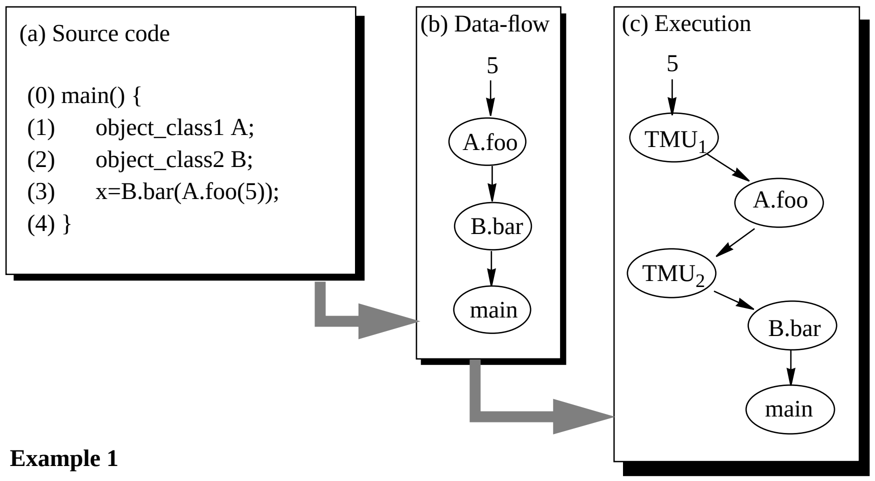
2.1. Mentat Execution Model

The Mentat execution model is based on the macro data-flow model (MDF [10]), an extension of the pure data-flow model. MDF is one of several large grain data flow models [3][6][8] that expand on traditional data flow. The salient features of MDF are that it incorporates the notion of state, adds the ability to dynamically create graphs, and provides coarse grained actors. In MDF, actors with states are said to be *persistent actors* while stateless actors are called *regular actors* (persistent and regular actors are implemented by persistent and regular Mentat objects).

The Mentat run-time system implements a virtual macro data-flow machine that transparently constructs program data-flow graphs, schedules actors on processors, and manages communication and synchronization. The token matching unit (TMU) implements the pure data-flow subset of the MDF model and is responsible for matching tokens and for enabling an actor when all its tokens are present. When all of the tokens required for an actor computation have arrived, the TMU issues a scheduling request to the system scheduler. The scheduler selects a processor to service the request and notifies the TMU, which then forwards the tokens to the actor so that it may execute. To distribute the workload associated with regular actors, there is one TMU per host and work is divided among them via a simple hash function.

The mapping from a sample MPL source code fragment that uses regular objects to the implementation is shown in Figure 2. At run-time, calls made to regular object functions (actors) are transformed into a data-flow graph (Figure 2b), which is then acted on by the run-time system to deliver the proper arguments (tokens) to the appropriate object's function (Figure 2c). For more information on the Mentat system, including how it detects data dependence and builds data-flow graphs, see [10][12].

Figure 2 Transformation from source code to execution



Example 1

First the graph is constructed at run-time in the main program. The data-flow graph is mapped onto the implementation as follows:

- A message containing the token "5" is sent to a TMU (TMU_1 in Figure 2c). The token contains a computation tag[1] that uniquely identifies the actor and the number of tokens required to enable the actor. The message also contains a copy of the program graph.
- Upon receiving the token, TMU_1 determines that the actor (A.foo()) may fire because all necessary tokens for the actor have arrived. TMU_1 makes a scheduling request to the system scheduler which creates an instance of object A and returns A's physical address (host id and port number).
- TMU_1 forwards the tokens and computation tags to object A.
- Object A executes function foo().
- When A.foo() finishes, it must send the result along all outgoing arcs in the program graph representation. Since B is a regular object, the result is passed to a TMU that is responsible for matching B.bar()'s tokens.
- B.bar() is handled similarly with the end results sent back to the main program.

1. Computation tags are similar to token colors [21].

3. Extending the model to support fault-tolerance

To provide transparent fault-tolerance we extend the run-time system to automatically replicate regular object execution. We first describe the interface and show that users can easily achieve fault-tolerance. We then describe the implementation of the replication protocol.

3.1. User interface

Users specify the degree of replication by creating an instance of the class `replication_manager` and setting the level of replication desired. The level of replication is valid within the scope of the declaration. The interface for `replication_manager` is shown below:

```
class replication_manager {
    public:
        void set_num_replicates(int howmany);
        replication_manager(howmany);
        ~replication_manager();
};
```

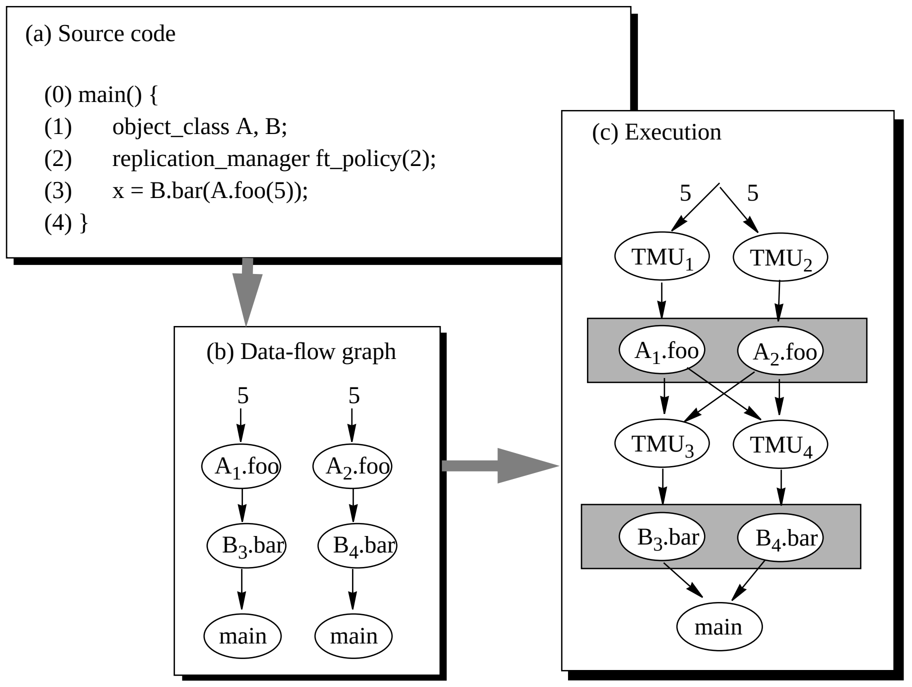
The constructor for the `replication_manager` class saves the current replication setting. Users specify a new setting within the constructor and can override their initial setting via the method `set_num_replicates()`. The new setting is then valid until the flow of control exits the current scope, at which point the destructor is implicitly invoked and restores the saved setting. The following code fragment illustrates the use of `replication_manager`:

```
main()
{
    int a;
    object_class X, Y;
    replication_manager ft_policy1(3); // replication level is 3

    a = X.op1 (Y.op2());
    {    // new scope
         int b;
         replication_manager ft_policy2(2); // replication level is now 2

         b = X.op1 (Y.op2());
    }
    // replication level is restored to 3
}
```

3.2. Implementation

The Mentat run-time system transparently replicates regular objects by duplicating the tokens and sending them to distinct TMUs. The choice of the additional TMUs is based on a hash function which is guaranteed to select distinct TMUs. However, naive duplication of tokens will lead to an exponential growth of tokens and computations. If we consider the data-flow graph in Figure 3b, naive replication would duplicate the “5” token and two A.foo actors would be created. Similarly, if the result from each A.foo actor were duplicated, four B.bar’s would be executed. For larger graphs this exponential growth would quickly overwhelm the system. To avoid exponential growth each TMU tracks the tokens it has already received. When, under normal circumstances, the duplicate tokens for a computation arrive, the TMU discards them. By discarding duplicates we avoid instantiating extra duplicate computations.

Figure 3 Implementing replication of regular objects



Example 2

We modified the program of example 1 in Figure 2 to set the replication level to two (Figure 3a). Execution proceeds as follows (Figure 3c):

- A message is created that contains the token “5”, the program graph, and the level of replication. This message is duplicated and sent to two distinct TMUs. The TMUs are chosen based on the computation tag for A.foo() using the primary and secondary hash function.
- Each TMU independently instantiates a copy of object A using the protocol described in Section 2.1. The actor corresponding to A.foo() is thus replicated.
- The result from each A.foo() is forwarded to each of the TMUs handling the next replicated actor (B.bar()). The TMU has been modified to detect duplicate tokens and discards the duplicates to prevent an exponential growth of objects instantiated.

One critique of this technique is that one cannot know how long to “remember” which tokens have already been consumed. In fact this is not a problem. We use a fixed size table of past tokens. When a new slot is needed we throw away the token with the oldest timestamp. In the unlikely event that a duplicate token arrives after we have “forgotten” about it we simply schedule a redundant computation whose result will be thrown away later.

Only minor changes to the TMU were required. No coordination is required between replicated objects nor between TMUs. The failure of any *k-1* of the TMUs handling a *k*-replicated actor, or *k-1* of the replicated objects, does not prevent the successful completion of the program, though a current requirement is that the host where the main program is running does not fail. Finally, the replication algorithm is decentralized and hence scalable.

The TMUs responsible for a given replicated object, i.e. TMU_1 and TMU_2 or TMU_3 and TMU_4, (Figure 3) are placed on distinct hosts by the Mentat system. Mentat does not currently guarantee that the replicated objects themselves execute on separate hosts, though this is likely to be the case. Assuming a random placement of objects and one host failure, the probability that all objects are placed on the failed host is given by: $P(n, z) = \frac{1}{z^n}$ where n is the number of replicates and z the total number of hosts. Under saturation, the Mentat scheduler [11] effectively uses a random placement policy.

4. Synthetic Pipeline Application

To explore the trade-offs between fault-tolerance, CPU resource consumption, and performance, we have created a synthetic pipeline application. In this application, the work is divided into independent pieces that flow through a two-stage pipeline with the results collected by the main program.

We have implemented two versions. The first version is a non fault-tolerant version using standard Mentat code and the non-enhanced run-time system. The second version is fault-tolerant employing the transparent replication method. Section 4 describes the implementation of each version in more detail while Section 5 discusses our performance results and demonstrates the trade-offs between fault-tolerance, resource consumption and performance.

4.1. Original MPL version

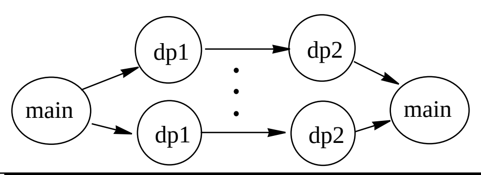
The non fault-tolerant version and its data-flow graph representation are shown in Figure 4. On line 3, we declare two regular objects that are the data processing filters. The number of iterations in the pipeline is `NUM_RESULTS`. The data flows through the two filters in a pipeline fashion before being stored in an array back in the main program. The data-flow graph generated from the source code implicitly shows the independence of the computations. Thus, the equivalent of a DO_ALL loop is automatically achieved.

Figure 4 Original source code and data-flow graph

(a) Source code for original non fault-tolerant Mentat version

```
(0) main() {
(1)      my_data the_data[NUM_RESULTS];
(2)      my_results result[NUM_RESULTS];
(3)      data_processor dp1, dp2;          // these are regular objects
(4)      for (i = 0; i < NUM_RESULTS; ++i)
(5)          result[i] = dp2.process(dp1.process(the_data[i]));
(6) }
```

(b) Data-flow graph representation



4.2. Transparent Replication Version

For the transparent replication method, the code is identical to the original code except for the specification of the replication policy on line 4 (Figure 5). The number of replicates is related to the level of fault-tolerance desired. By default, all regular objects are 1-replicated. In general, a *k*-replicated object will tolerate k-1 failures assuming that all objects are placed on a different host. Setting the number of replicates high will improve the fault-tolerant characteristics of the application at the cost of higher resource consumption and possibly worse performance (depending on the number of hosts available).

Figure 5 Source code for transparent replication method

```
(0) main() {
(1)      my_data the_data[NUM_RESULTS];
(2)      my_results result[NUM_RESULTS];
(3)      data_processor dp1, dp2;
(4)      replication_manager ft_policy(number_of_replicates);
(5)
(6)      for (i = 0; i < NUM_RESULTS; ++i)
(7)          result[i] = dp2.process(dp1.process(the_data[i]));
(8) }
```

5. Results

When evaluating a fault-tolerance mechanism for a parallel computing environment we must keep firmly in mind that performance is the *raison d'être* of parallel computing. In a shared computing environment with multiple users, resource consumption is also an issue; any resources used to ensure fault-tolerance for one application cannot be used by another application. A design that incurs high overhead, recovers failed computations slowly, or uses large amounts of resources will not be used if the price is too high.

To determine the relative strengths and weaknesses of the transparent replication method, we tested its performance with a synthetic pipeline application and used the non-fault tolerant implementation as a baseline. We tested the transparent replication method under no failure and single failure scenarios to determine its recovery time characteristics. We tested each configuration on a variety of workloads, ranging from 1-32 pipeline iterations with each stage of the pipe taking approximately 13 seconds[4]. The times presented below are the average start-to-finish wall clock times over 25 runs on a dedicated network of 8 Sun SparcStation2 workstations.

5.1. Non Fault-tolerant Baseline Case

In Table 1 we show the average wall clock time elapsed for the baseline case with no failures. Notice that for 1 to 4 iterations of the pipe performance remains nearly constant. This is because Mentat automatically detects the independence of each iteration of the main loop and immediately schedules the first stage of the pipeline across all iterations. The theoretical limit for the pipeline is reached with 8 iterations. In practice, this limit is often not achieved because the scheduler may place multiple objects on the same host and thus the performance degrades as the number of iterations increases to 8.

5.2. Transparent Replication

An advantage of the transparent replication method is that it essentially provides instantaneous recovery in the presence of failure. Moreover, it is generic and easy to use. Applications are not limited to a particular structure, as the Mentat run-time system can handle arbitrarily complex data-flow graphs. The programmer sets the replication level for objects and does not need to worry about the fault-tolerance protocols involved. The main drawback of this method is its high usage of CPU resources.

Table 1 shows performance and resource consumption with the level of replication set to 2. Resource consumption is directly related to the replication level and thus twice as many resources are consumed. When the available set of hosts is saturated with objects, performance decreases in relation to the non fault-tolerant baseline case as replicates delay the execution of other objects. This effect can be seen especially for 16 and 32 iterations, where performance respectively degrades by 28% and 55%. On the other hand, when the number of hosts exceeds the number of objects, there is almost no performance penalty.

---

[4]. There is nothing special about 13 seconds. In fact, the granularity could be smaller by two orders of magnitude.

Table 1. Performance and CPU resources consumed with 0 & 1 host failure simulated

	No host failure				1 host failure	
	Baseline		2-replicated		2-replicated	
Iterations	Time (sec)	Total Resources (CPU sec)	Time (sec)	Total Resources (CPU sec)	Time (sec)	Total Resources (CPU sec)
1	27	26	26	52	28	50
2	28	52	27	104	29	95
4	33	104	36	208	39	196
8	50	208	58	416	58	370
16	71	416	91	832	93	770
32	120	832	186	1664	191	1540

There is no significant difference in performance between the 0 and 1 host failure case with the replication level set to two. This is expected since the objects that were placed on the failed host have a duplicate on another host. Resource consumption is slightly lower with one host failure as the objects that are placed on the failed host only partially execute or do not execute at all.

6. Dormant replicates

In the previous replication technique, replicated actors were always active - they fired as soon as their tokens were matched. The result was a profligate use of resources — even when there was no failure. To address the resource consumption problem we now introduce the concept of dormant actors. A dormant actor does not fire immediately when enabled. Instead, its TMU monitors the state of an active actor by periodically sending ping messages. When the active actor fails to respond within some specified time interval, the TMU reissues the actor computation.

The combination of active and dormant actors affords users the flexibility of controlling the resource consumption and aggressiveness of their replication policy. By using dormant actors, users may reduce the consumption of CPU resources while arbitrarily increasing the level of fault-tolerance. However, they may pay a performance penalty in the presence of failures as time would be lost in (1) the partial work already performed by the failed object, (2) detecting failure, and (3), restarting and re-executing the computation. By judiciously choosing the number of active actors, the number of dormant actors and their associated ping intervals, users may maximize their objective function, whether it be to achieve the best performance, the efficient use of CPU resources or a combination of both. This approach does not enforce a particular policy but leaves that decision in the hands of the application writer.

6.1. Interface for specifying active and dormant actors

Users specify the replication policy by creating an instance of a new `replication_manager` class and setting the number of active actors, the number of dormant actors and their ping intervals. The interface is shown below:

```
class replication_manager {
    public:
        void set_num_active(int howmany);
        void set_num_dormant(int howmany, int pingInterval);
        void set_num_dormant(int howmany, int pingIntervals[]);
        // constructors
        replication_manager(int numActive);
        replication_manager(int numActive, int numDormant, int pingInterval);
        replication_manager(int numActive, int numDormant, int pingInterval[]);
        ~replication_manager();
};
```

Note that there are two ways of setting the ping interval. In the first, users specify the ping interval in seconds for all dormant objects. In the second, users specify the ping interval for each dormant object. Figure 6 illustrates the use of the `replication_manager` class.

6.2. Implementation

The improved replication protocol is much more complex and requires changes to both TMUs and regular objects. Replication is still achieved by duplicating tokens and sending them to distinct TMUs, except that now, the level of replication is specified by summing the number of active and dormant replicates. Moreover, we require at least one active object. To clarify the description of the new protocol, we distinguish between active and dormant TMUs, i.e. TMUs that handle active objects or dormant objects respectively[5]. We now describe the protocol for both kind of TMUs.

Recall that in the original protocol TMUs make a scheduling request and then immediately forward the tokens to the newly instantiated object. In the new protocol, active TMUs must also send the object's name to dormant TMUs so that the latter can periodically ping the object. Dormant TMUs do not make a scheduling request when the tokens are matched but instead start monitoring the enabled actor by using the ping interval P associated with the actor. If within P seconds, the dormant TMU has not received the name of the active object from an active TMU, it will go ahead and schedule another instance of the actor. Otherwise, the dormant TMU pings its associated active object every P seconds. If the active object fails to respond within P seconds, the dormant TMU considers it dead and reschedules another instance.

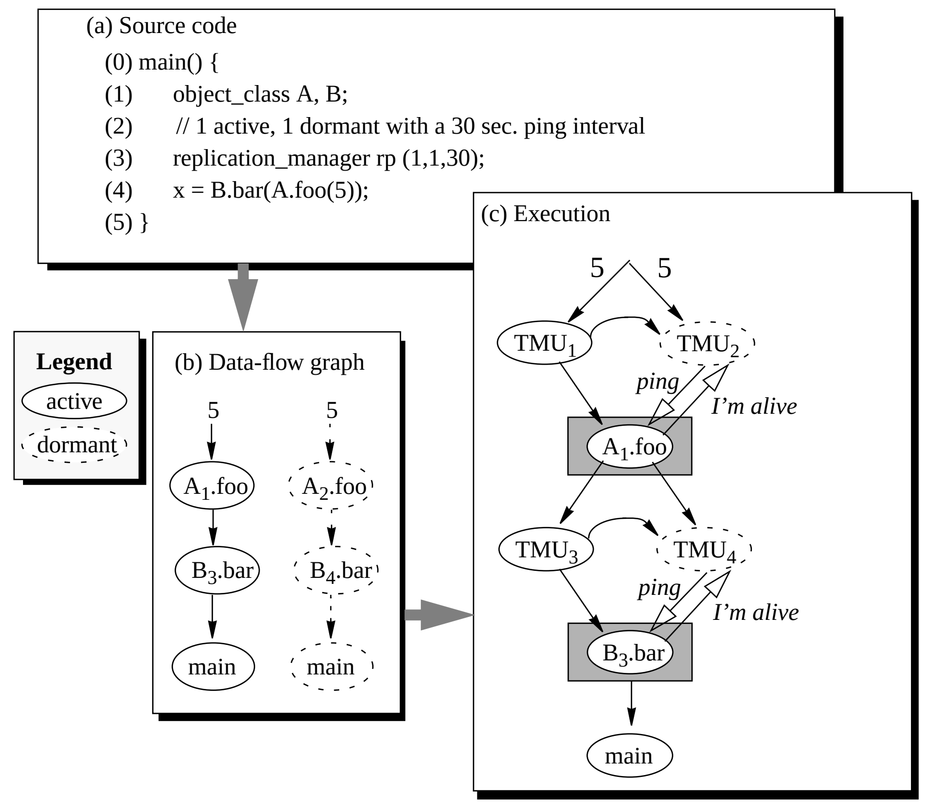
The implementation of regular objects must also change. In addition to forwarding their results to the next nodes in the data-flow graph, regular objects must also notify all

[5]. TMUs may be both active and dormant — they are active or dormant with respect to a particular actor.

dormant TMUs that they have finished their computation. Otherwise, dormant replicates would fire within at most P seconds of completion of the active replicate. All dormant TMUs associated with the finished computation can then stop monitoring the status of replicated active objects and discard the unused tokens.

Note that we define failure as the inability to respond. The active object may in fact be alive but may be unable to respond because of a network failure, a processor overload, or some other problems. While we may schedule a redundant computation as a result of our definition of failure, the algorithm is still correct since actors may be safely replicated due to their idempotent nature. Of course, the downside of false failure is unnecessarily increased resource consumption.

Figure 6 Replication with active and dormant regular objects


(a) Source code

```
main() {
      object_class A, B;
      // 1 active, 1 dormant with a 30 sec. ping interval
      replication_manager rp (1,1,30);
      x = B.bar(A.foo(5));
}
```

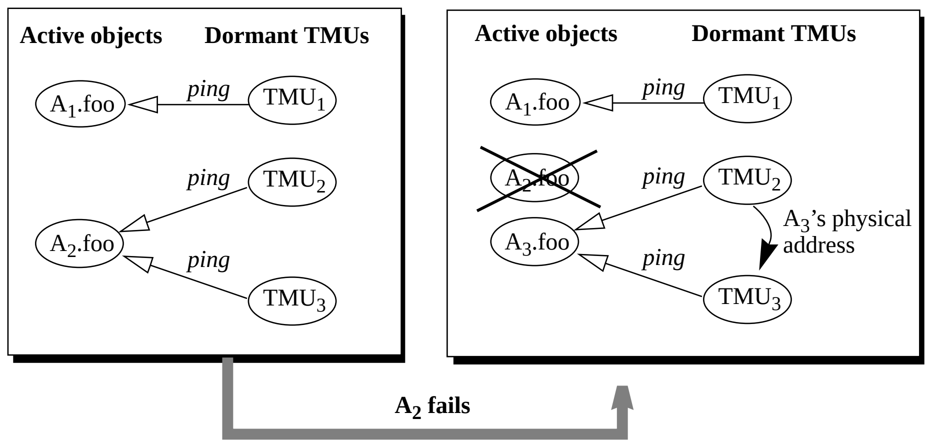
Active TMUs use a simple round-robin scheme to select their associated dormant TMUs[6]. A possible result of this assignment is that an active object may be monitored by more than one dormant TMU. If the active object were to fail, then multiple copies of the object would be started. To alleviate this problem, the first dormant TMU to

[6]. TMUs are ordered by using the same hash function on the computation tag.

restart the failed computation informs the other dormant TMUs of the newly activated object's physical address. Consider the scenario in Figure 7 where the replication policy is set to 2 active and 3 dormant objects. TMU_2 detects A_2's failure, restarts the computation (A_3.foo) and sends TMU_3 A_3's physical address. TMU_3 then pings A_3 instead of the failed A_2.

Even with this algorithm, it is still possible to unnecessarily instantiate multiple copies of a computation. For example, TMU_2 and TMU_3 could have simultaneously detected A_2's failure and independently restarted the computation. However, this is a worst case scenario that only affects the resources consumed and not the correctness of the result.

Figure 7 Dormant TMUs sharing the same active object



7. Related work

While there is a rich literature in fault-tolerance for distributed and real-time systems (see for examples the proceedings of Fault-Tolerant Computing Symposium (FTCS) and Real-Time Operating Systems (RTOS)), there has been much less done in the area of fault-tolerant parallel processing systems. Most of the work has concentrated on fault-tolerant hardware, e.g. fault-tolerant networks and system reconfiguration after a fault. There has been some though, for example, FT-Linda [4], PLinda [15], Orca [16], Calypso [5], and Fail-safe PVM [17]. These systems use a combination of well known mechanisms such as replication, transactions, message logging, or checkpoints and rollbacks to provide fault-tolerance.

Mentat differs from these systems in that its underlying computational model is based on data-flow. Moreover, Mentat and macro data-flow (MDF) differ from other large grained data-flow systems such as Paralex [2], CDF [3], HeNCE [6], and Code/Rope [8] in that program graphs in MDF are dynamic and generated at runtime. In Mentat, the program graphs are generated by the compiler and run-time system, unlike [2][6][8], where the programmer is responsible for generating the program graphs using a graphical interface. Paralex uses the ISIS toolkit to provide fault-tolerance via the coordinator-cohort model [7]. To our knowledge Paralex is one of the few data-flow parallel processing system that provides direct support for fault-tolerance. ATAMM [19] is another but its application domain is embedded real-time systems.

The techniques described in this paper are easily applicable to any coarse grain data-flow systems. However, replication is not novel and is a well understood concept even in the general case of objects/processes with state [18][20]. Our work differs in that we have focussed on the special case, i.e. stateless objects, and exploited their idempotent nature to provide easy-to-use fault-tolerance. Further, the replication protocol is greatly simplified as the system does not need to maintain consistency between replicates or take checkpoints and rollback.

8. Conclusion

Wide-area parallel processing systems will soon be available to researchers to solve a range of problems. It is certain that host failures and other faults will be an every day occurrence in these systems. Unfortunately contemporary parallel processing systems were not constructed with fault-tolerance as a design objective.

The data-flow model, long a mainstay of parallel processing, offers hope. The model's functional nature, which makes it so amenable to parallel processing, also facilitates straight-forward fault-tolerant implementations. It is the combination of ease of parallelization and fault-tolerance that we feel will increase the importance of the model in the future, and lead to the widespread use of functional components.

To illustrate our point, we have modified the Mentat run-time system to provide transparent replication of data-flow actors. The advantages of this method are that it is easy to use, programmers simply set the level of replication desired in the parts of their program that need fault-tolerance, and generic, it works with arbitrarily complex data-flow graphs. Its main drawback is the high consumption of CPU resources. Furthermore, we have found that while setting the level of replication high can improve the fault-tolerance characteristics of an application, it can also have adverse effects on performance. When hosts are saturated with objects, performance decreases as replicated objects compete with other objects for CPU resources. However, we have outlined an optimization that can significantly reduce the amount of CPU resources consumed while giving users a flexible interface for controlling the replication policy.

Using Mentat, programmers have at their disposal a “dial” with which to trade-off fault-tolerance, performance and resource consumption. Where programmers choose to set the “dial” ultimately depends on the relative importance that they attach to fault-tolerance, performance and resource consumption.

Now that we have demonstrated that robust techniques can be added to our existing system, the next steps are to implement the optimized replication method and investigate its performance and resource consumption characteristics with a diverse set of applications on the University of Virginia's campus-wide virtual computer. In addition, we plan to provide mechanisms to support fault-tolerance for persistent objects.

9. Acknowledgments

We would like to thank Adam Ferrari and Mark Hyett for their suggestions in writing this paper. We would also like to thank the reviewers for their insightful comments on an earlier draft of the paper.

10. References

[1] T. Agerwala and Arvind, “Data Flow Systems,” *IEEE Computer*, vol. 15, no. 2, pp. 10-13, February, 1982.

[2] O. Babaoglu et. al., “Paralex: An Environment for Parallel Programming in Distributed Systems,” Technical Report UBLCS-92-4, Laboratory for Computer Science, University of Bologna, Oct. 1992.

[3] R. F. Babb, ”Parallel Processing with Large-Grain Data Flow Techniques,” *IEEE Computer*, pp. 55-61, July, 1984.

[4] D. Bakken and R. Schlichting, “Supporting fault-tolerant parallel programming in Linda,” Technical Report TR93-18, The University of Arizona, 1993.

[5] A. Baratloo, P. Dasgupta and Z. M. Kedem, “CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms,” *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pp. 122-129, Washington, D.C., August 1995.

[6] A. Beguelin et al., “HeNCE: Graphical Development Tools for Network-Based Concurrent Computing,” *Proceedings SHPCC-92*, pp. 129-136, Williamsburg, VA, May, 1992.

[7] K. Birman et. al., “Implementing Fault-Tolerant Distributed Objects,” *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, June 1985.

[8] J. C. Browne, T. Lee, and J. Werth, “Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment,” *IEEE Transactions on Software Engineering*, pp. 111-120, vol. 16, no. 2, Feb., 1990.

[9] A. S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, pp. 39-51, May, 1993.

[10] A. S. Grimshaw, “The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing,” Computer Science Technical Report, CS-93-30, University of Virginia, May, 1993.

[11] A. S. Grimshaw and V. E. Vivas, “FALCON: A Distributed Scheduler for MIMD Architectures”, *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 149-163, Atlanta, GA, March, 1991.

[12] A. S. Grimshaw, J. B. Weissman and W. T. Strayer, “Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing”, *To appear in the ACM Transactions of Computer Systems*.

[13] A. S. Grimshaw, A. Nguyen-Tuong and W. A. Wulf, “Campus-Wide Computing: Early Results using Legion at the University of Virginia”, Technical Report CS-95-19, Department of Computer Science, University of Virginia, 1995.

[14] A. S. Grimshaw et. al., “Legion: The Next Logical Step Toward a Natiowide Virtual Compute,” Computer Science Technical Report, CS-94-21, June 8, 1994.

[15] K. Jeong and D. Shasha, “Plinda 2.0: A transactional/checkpointing approach to fault tolerant Linda,” *Proceedings of the 13th Symposium on Reliable Distributed Systems*, 1994.

[16] M. Kaashoek et. al., “Transparent fault-tolerance in parallel Orca programs,“ *Symposium on Experiences with Distributed and Multiprocessor Systems*, 1992.

[17] J. Leon, A. L. Fisher, P. Steenkiste, “Fail-safe PVM: A portable package for distributed programming with transparent recovery”, Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, PA, February 1993.

[18] M.C. Little and S.K. Shrivastava, “Replicated K-Resilient Objects in Arjuna”, P*roceedings of the 1st IEEE Workshop on the Management of Replicated Data*, Houston, pp. 53-58, November 1990.

[19] R. A. Obando and J. W. Stoughton, “A Performance Prediction Model for a Fault-Tolerant Computer During Recovery and Restoration,” NASA Contractor Report 195074, NASA Langley Research Center, Virginia, February 1995.

[20] D. Powell, “Delta-4: A Generic Architecture for Dependable Distributed Computing,” ESPRIT project 2252 Research Report, Springer Verlag, 1991.

[21] A. H. Veen, “Dataflow Machine Architecture,” *ACM Computing Surveys*, pp. 365-396, vol. 18, no. 4, December, 1986.